第６章　MPI_c ライブラリーの一覧

6.1 はじめに

本章では、MPI_c ライブラリーに関する全プログラムを掲載する。このライブラリーのファイル構成は以下のようであり、この順番で、ライブラリーの内容を説明する。

1）	MPI_C_Core.h	MPI_C_Core.cpp
2）	mpi_c.h	mpi_c.cpp
3）	sock_func.h	sock_func.cpp
4）		mpi_func.cpp
5）	mpi_c.def	

6.2 MPI_C_Core

ここでは､MPI_C_Core クラスに関するヘッダーファイルと本文を示す。最初は、ヘッダーファイル MPI_C_Core.h であり、このクラスのメンバー関数とメンバー変数が定義されている。メンバー関数の中で、Send()と Recv()は、インラインで定義されている。

```
// MPI_C_Core.h: CMPI_C_Core クラスのインターフェイス
//
//////////////////////////////////////////////////////////////////////
#if !defined(AFX_MPI_C_CORE_H__1C6F0D42_BEFB_4CC1_B2FF_AAF2169ED5F4__INCLUDED_)
#define AFX_MPI_C_CORE_H__1C6F0D42_BEFB_4CC1_B2FF_AAF2169ED5F4__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <winsock2.h>
#include "sock_func.h"
#include "mpi_c.h"

class CMPI_C_Core
{
public:
	CMPI_C_Core();
	virtual ~CMPI_C_Core();
	// 初期化・終了
	bool Init(int nID, const char* szParam);		// mpi_cの初期化処理
	void ShutDownAll();					// 全ての接続を切断する
	// 送受信
	inline int Send(int nID, const void* pBuf, int nSize);	// データの送信を行う
	inline int Recv(int nID, void* pBuf, int nSize);	// データの受信を行う
	int BCast(int nRoot, void* pBuf, int nSize);		// データの一斉送信を行う
```

```
        // ファイル読み込み
        bool ReadProcess(const char* szPass);                // プロセス設定ファイルの読み込みを行う
        // 状態取得
        int GetProcessCount() const;                         // 起動しているプロセスの数を取得
        int GetMyID() const;                                 // 自分のプロセス番号を取得
        int GetRange_S(int nID) const { return m_pInfo[nID].m_nRange[0]; }  // 指定IDの担当範囲(開始)を取得
        int GetRange_E(int nID) const { return m_pInfo[nID].m_nRange[1]; }  // 指定IDの担当範囲(終了)を取得
        PROCESS_INFO GetInfo(int i) { return m_pInfo[i]; } // 指定IDのプロセス情報を取得
        // 内部初期化処理
private:
        bool SendBoot(SOCKET hSocket, int nID);      // デーモンに起動要求を送信する
        bool SendProcessInfo(SOCKET hSocket);        // プロセス情報を送信
        bool RecvProcessInfo(SOCKET hSocket);        // プロセス情報を受信
        // オブジェクト
public:
        SOCKET       m_pSocket[MAX_PROCESS];         // ソケットハンドル[]
private:
        PROCESS_INFO m_pInfo[MAX_PROCESS];           // プロセス情報[]
        int          m_nProcess;                     // プロセス数
        int          m_nMyID;                        // 自分のID
        char   m_szSlave[PARAM_SIZE];                // スレーブの実行ファイル名(拡張子も含む)
        // 計測用
public:
        int Time_Start(int i = 0);                   // 時間の計測を開始
        double Time_Stop(int i = 0);                 // 計測時間を取得
        bool TimeLog(const char *szLogFile);         // 計測ログを記録する
private:
        LONGLONG     m_llFreq;                       // カウンタ周波数
        LONGLONG     m_llCnt[MAX_TIMER][2];          // 時間
        LONGLONG     m_llAllCnt[MAX_TIMER];          // 合計時間
};

int CMPI_C_Core::Send(int nID, const void* pBuf, int nSize)
{
        return sock_send(m_pSocket[nID], pBuf, nSize);
}

int CMPI_C_Core::Recv(int nID, void* pBuf, int nSize)
{
        return sock_recv(m_pSocket[nID], pBuf, nSize);
}

#endif // !defined(AFX_MPI_C_CORE_H__1C6F0D42_BEFB_4CC1_B2FF_AAF2169ED5F4__INCLUDED_)
```

次に、MPIC_C_Core クラスを以下に示す。ここでは、各メンバー関数について説明するが、関数内では、Socket 関数が使用されている。

```
// MPI_C_Core.cpp: CMPI_C_Core クラスのインプリメンテーション
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
```

```
#include "mpi_c.h"
#include "sock_func.h"
#include "MPI_C_Core.h"

#include <fstream.h>

//////////////////////////////////////////////////////////////////////
// 構築/消滅
//////////////////////////////////////////////////////////////////////

CMPI_C_Core::CMPI_C_Core()
{
	int i;
	for(i=0; i < MAX_PROCESS; i++)
	{
		// ソケットハンドル[]の初期化
		m_pSocket[i] = INVALID_SOCKET;
		// プロセス情報[]の初期化
		strset(m_pInfo[i].m_szHost, 0);
		m_pInfo[i].m_nID = -1;
		m_pInfo[i].m_nRange[0] = -1;
		m_pInfo[i].m_nRange[1] = -1;
	}
	m_nProcess   = 0;		// プロセス数
	m_nMyID		= -1;		// 自分のID
	// Time_Start,Time_Stop,TimeLogで用いられるタイマーの初期化
	int r;
	// CPUのカウンタ周波数を取得する
    r = QueryPerformanceFrequency( (LARGE_INTEGER *)&m_llFreq );
	// タイマーカウンタの初期化
	for(i=0; i < MAX_TIMER; i++)
	{
		m_llCnt[i][0] = 0;
		m_llCnt[i][1] = 0;
		m_llAllCnt[i] = 0;
	}
}

CMPI_C_Core::~CMPI_C_Core()
{
	ShutDownAll();				// 全てのソケットを切断する
}

void CMPI_C_Core::ShutDownAll()
{
	int i;
	if(m_pSocket)
	{
		for(i=0; i < m_nProcess; i++)
		{
			sock_close(m_pSocket[i]);	// ソケットを切断する
			m_pSocket[i] = INVALID_SOCKET;	// ソケットを無効な値とする
		}
	}
```

```
	// Winsockの解放を行う
	WSACleanup();
}

// ================================================================
// CMPI_C_Core::Init
// 概要　: mpi_cの初期化を行う。プロセス情報を元に全てのプロセスを接続を行う。
//		初期化の基本的な流れは以下の通りである。
//		①プロセス情報の取得
//			master : 設定ファイルを読み込む
//			slave  : 一つ前のプロセスと接続し、プロセス情報を受け取る。
//		②接続アルゴリズム
//			自分より小さいID		: 接続する
//			自分より一つ大きいID	: 起動させ、接続を待ち受ける
//			自分より大きいID		: 相手からの接続を待ち受ける
//	※　起動させるプロセスではmpi_cデーモンが起動している必要がある。
// 引数　:　nID		= 自分のID
//		szParam		= パラメータ引数
//				master: プロセス設定ファイルのフルパス,
//				slave: 一つ前のIDのコンピュータ名(接続用に用いる)
// 戻り値: bool	: 他のプロセスとの接続に失敗した場合はfalseが返る
// ================================================================
bool CMPI_C_Core::Init(int nID, const char* szParam)
{
	int i;			// カウンタ
	SOCKET hDeamon;		// デーモン接続用ソケット
	SOCKET hSocket;		// ソケットの一時保存用
	// Winsockの初期化
	if(sock_init() == false)
	{
		return false;
	}
	// マスター(nID = 0)なら、プロセス設定ファイルを読み込む
	if(nID == ID_MASTER)
	{
		ReadProcess(szParam);

	}
	// それ以外なら、自分の前のプロセスにアクセスして、プロセス情報を受け取る
	else
	{
		hSocket = sock_connect(szParam, PORT_PARALLEL + (nID -1));
		if(hSocket == INVALID_SOCKET)	return false;
		RecvProcessInfo(hSocket);
	}
	// 自分のIDを設定
	m_nMyID = nID;

	// 接続開始
	for(i=0; i < m_nProcess; i++)
	{
		// 自分のひとつ前のプロセスは無視(既に接続しているので)
		if(i == m_nMyID -1)
		{
```

```
                m_pSocket[i] = hSocket;
            }
            // 自分より小さいプロセスには接続
            else if(i < m_nMyID)
            {
                m_pSocket[i] = sock_connect(m_pInfo[i].m_szHost, PORT_PARALLEL + i);
                if(m_pSocket[i] == INVALID_SOCKET)
                {
                    return false;
                }
            }
            // 自分の場合は待ち受け開始
            else if(i == m_nMyID)
            {
                // クライアントからの待ち受け用ソケットを作成
                m_pSocket[i] = sock_listen(PORT_PARALLEL + m_nMyID);
                if(m_pSocket[i] == INVALID_SOCKET)
                {
                    return false;
                }
            }
            // 自分より１つ大きいプロセスは起動させて、接続を待つ
            else if(i == m_nMyID + 1)
            {
                // デーモンと接続し、起動要求を出す
                hDeamon = sock_connect(m_pInfo[i].m_szHost, PORT_DEAMON);
                SendBoot(hDeamon, i);
                sock_close(hDeamon);
                // 自分より１つ大きいプロセスからの接続を待つ
                if(sock_accept(m_pSocket[m_nMyID], m_pSocket[i]) == false)
                {
                    return false;
                }
                // プロセス情報を送信
                SendProcessInfo(m_pSocket[i]);
            }
            // 自分より大きいプロセスは接続を待つ
            else
            {
                if(sock_accept(m_pSocket[m_nMyID], m_pSocket[i]) == false)
                {
                    return false;
                }
            }
        }
        return true;
    }

    // ================================================================
    // CMPI_C_Core::ReadProcess
    // 概要　: mpi_cのプロセス設定ファイルの読み込みを行う。
    //                設定ファイルはACIIファイルで以下の書式でなければならない。
    //                書式が誤っている場合の動作は不定である。
    //          ①[起動させるプロセスの実行ファイル名.exe]
```

```
//          ②[プロセスを実行するコンピュータ名(IPアドレス)] [担当範囲(開始)] [担当範囲(終了)]
//          最初の行に①を記入し、以下、設定したいプロセスの分だけ②を繰り返す。
//          プロセスのIDは記入された順にID=0,1,2,3,...となる。
//          ※ ②には最初の行はマスターでなければならない。つまり、ID=0として
//                初期化されるプロセスが実行されているコンピュータの名前でなければならない。
// 引数   : szPass  = プロセス設定ファイルのフルパス
// 戻り値: bool     : ファイルが開けなかった場合はfalseを返す
// ===================================================================
#include <vector>
bool CMPI_C_Core::ReadProcess(const char* szPass)
{
      PROCESS_INFO info;
      std::vector<PROCESS_INFO> vecInfo;
      int i;
      // ファイルを開く
      ifstream file(szPass, ios::in);
      if(!file)    return false;
      file >> m_szSlave;
      // プロセス設定情報を読み込む
      while(file >> info.m_szHost >> info.m_nRange[0] >> info.m_nRange[1])
      {
            vecInfo.push_back(info);
      }
      // プロセス情報オブジェクトを生成
      m_nProcess = (int)vecInfo.size();
      if(m_nProcess > MAX_PROCESS)    m_nProcess = MAX_PROCESS;

      for(i=0; i < m_nProcess; i++)
      {
            strcpy(m_pInfo[i].m_szHost, vecInfo[i].m_szHost);
            m_pInfo[i].m_nID = i;
            m_pInfo[i].m_nRange[0] = vecInfo[i].m_nRange[0];
            m_pInfo[i].m_nRange[1] = vecInfo[i].m_nRange[1];
      }
      return true;
}

// ===================================================================
// CMPI_C_Core::SendBoot
// 概要   : デーモンにプロセス起動要求を送る。
//                デーモンの起動しているフォルダにCMPI_C_Core::m_szSlaveに
//                記されている実行ファイルがある必要がある。
//          ※ 事前に、
//          sock_connect([プロセスを起動したいコンピュータ名], PORT_DEAMON)
//          で、デーモンと接続をしておく必要がある。
// 引数   :  hSocket      = デーモンと接続しているソケットのハンドル
//                nID          = 起動させたいプロセスにつけるID
// 戻り値: bool     : 接続要求の送信に失敗した場合はfalseを返す
// ===================================================================
bool CMPI_C_Core::SendBoot(SOCKET hSocket, int nID)
{
      int nRet;
      bool bRet;
      char szParam[PARAM_SIZE];
```

```
	// メッセージを送信
	int nBoot = MPI_MSG_BOOT;
	nRet = sock_send(hSocket, &nBoot, sizeof(nBoot));
	if(nRet < sizeof(nBoot))  return false;
	// パラメータ１([リモート起動させる実行ファイル名])を送信
	nRet = sock_send(hSocket, m_szSlave, sizeof(m_szSlave));
	if(nRet < sizeof(m_szSlave))    return false;
	// パラメータ２([起動させるプロセスのID] [自分のIPアドレス])を送信
	char szIP[MAX_COMPUTERNAME_LENGTH +1];
	bRet = get_local_IP(szIP, sizeof(szIP));
	if(bRet == false)  return false;
	sprintf(szParam, "%d %s", nID, szIP);
	nRet = sock_send(hSocket, szParam, sizeof(szParam));
	if(nRet < sizeof(szParam))      return false;
	return true;
}

// ================================================================
// CMPI_C_Core::SendProcessInfo
// 概要  : プロセス情報[]を送信する。
// 引数  :  hSocket      = プロセス情報を送信するソケットのハンドル
// 戻り値: bool    : プロセス情報[]の送信に失敗した場合はfalseを返す
// ================================================================
bool CMPI_C_Core::SendProcessInfo(SOCKET hSocket)
{
	int nRet;
	nRet = sock_send(hSocket, &m_nProcess, sizeof(m_nProcess));           // プロセス数
	if(nRet < sizeof(m_nProcess))   return false;
	nRet = sock_send(hSocket, &m_pInfo, sizeof(PROCESS_INFO)*m_nProcess); // プロセス情報
	if(nRet < (sizeof(PROCESS_INFO) * m_nProcess))     return false;
	nRet = sock_send(hSocket, m_szSlave, sizeof(m_szSlave));              // 起動アプリ名
	if(nRet < sizeof(m_szSlave))    return false;
	return true;
}

// ================================================================
// CMPI_C_Core::RecvProcessInfo
// 概要  : プロセス情報[]を受信する。
// 引数  :  hSocket      = プロセス情報を受信するソケットのハンドル
// 戻り値: bool    : プロセス情報[]の受信に失敗した場合はfalseを返す
// ================================================================
bool CMPI_C_Core::RecvProcessInfo(SOCKET hSocket)
{
	int nRet;
	nRet = sock_recv(hSocket, &m_nProcess, sizeof(m_nProcess));           // プロセス数
	if(nRet < sizeof(m_nProcess))   return false;
	nRet = sock_recv(hSocket, &m_pInfo, sizeof(PROCESS_INFO)*m_nProcess); // プロセス情報
	if(nRet < (sizeof(PROCESS_INFO) * m_nProcess))     return false;
	nRet = sock_recv(hSocket, m_szSlave, sizeof(m_szSlave));              // 起動アプリ名
	if(nRet < sizeof(m_szSlave))    return false;
	return true;
}

// ================================================================
```

```
// CMPI_C_Core::BCast
// 概要　: 全てのプロセスにデータを一斉送信する。
//           ※ CMPI_C_Core::BCast()を用いた場合、データの受信側でも必ず
//                 CMPI_C_Core::BCast()を用いることにより受信を行わなければならない。
//                 CMPI_C_Core::recv()で受信を試みた場合の動作は不定であるので
//                 注意が必要である。
// 引数　:   nRoot = 送信元プロセスのID
//                 pBuf   = 送信データの先頭アドレス(データ型は問わない)
//                 nSize  = 送信データのサイズ(sizeof()演算子で取得)
// 戻り値: int     : 一斉送信を行う側で失敗した場合、nSize以外の値が返る
//                       受信する側で失敗した場合は、途中まで受信したサイズが返る
// ===================================================================
int CMPI_C_Core::BCast(int nRoot, void* pBuf, int nSize)
{
      // 後でアルゴリズムを修正
      int nRet;
      int i;
      if(nRoot == m_nMyID)
      {
            for(i=0; i < m_nProcess; i++)
            {
                  if(i == nRoot)     continue;
                  nRet = Send(i, pBuf, nSize);
                  if(nRet < nSize)   return nRet;
            }
      }
      else
      {
            nRet = Recv(nRoot, pBuf, nSize);
            if(nRet < nSize)   return nRet;
      }
      return nRet;
}

// ===================================================================
// CMPI_C_Core::Time_Start
// 概要　: 時間の計測を開始する。独立したタイマーを保持しているので
//                 MAX_TIMER(mpi_c.hで定義)個を同時に計測することができる
// 引数　:   i     = 用いるタイマーの番号(0からMAX_TIMER-1までの間)
// 戻り値: int     : 不正な引数が渡された場合は-1が返る。それ以外は0が返る
// ===================================================================
int CMPI_C_Core::Time_Start(int i)
{
      if(i < 0 || i >= MAX_TIMER)     return -1;
      QueryPerformanceCounter((LARGE_INTEGER *)&m_llCnt[i][0]);
      return 0;
}

// ===================================================================
// CMPI_C_Core::Time_Stop
// 概要　: 時間の計測を終了し、計測した時間をマイクロ秒単位で返す。
//                 独立したタイマーを保持しているので
//                 MAX_TIMER(mpi_c.hで定義)個を同時に計測することができる
// 引数　:   i     = 用いるタイマーの番号(0からMAX_TIMER-1までの間)
```

```
// 戻り値: double   : 計測された時間(マイクロ秒)を返す。
// ===================================================================
double CMPI_C_Core::Time_Stop(int i)
{
	LONGLONG llDifCnt;
	if(i < 0 || i >= MAX_TIMER)	return -1;
	QueryPerformanceCounter((LARGE_INTEGER *)&m_llCnt[i][1]);
	llDifCnt = m_llCnt[i][1] - m_llCnt[i][0];
	m_llAllCnt[i] += llDifCnt;
	return (double)llDifCnt / m_llFreq * 1000000.0;
}

// ===================================================================
// CMPI_C_Core::TimeLog
// 概要   : 指定されたファイル名にCPUカウンタ周波数と
//                各タイマーのカウンタ値の総和を記録する。
// 引数   :   szLogFile    = 記録するファイル名
// 戻り値: bool     : ファイルのオープンに失敗すればfalseを返す
// ===================================================================
bool CMPI_C_Core::TimeLog(const char* szLogFile)
{
	ofstream file(szLogFile, ios::out);
	if(!file)	return false;
	file << "Freq = " << (unsigned long)m_llFreq << endl;
	for(int i=0; i < MAX_TIMER; i++)
	{
		file << "AllCnt[" << i << "]=" << (unsigned long)m_llAllCnt[i] << endl;
	}
	return true;
}

// ===================================================================
// CMPI_C_Core::GetProcessCount
// 概要   : mpi_cで起動しているプロセスの数を取得する。
// 引数   :   none
// 戻り値: int      : 起動しているプロセスの数(masterも含む)
// ===================================================================
int   CMPI_C_Core::GetProcessCount() const
{
	return m_nProcess;
}

// ===================================================================
// CMPI_C_Core::GetMyID
// 概要   : mpi_c内の自分のプロセスIDを取得する
// 引数   :   none
// 戻り値: int      : 自分のプロセスID(master:0, slave=1,2,...)
// ===================================================================
int CMPI_C_Core::GetMyID() const
{
	return m_nMyID;
}
```

最初は、MPI_c ライブラリーに関するヘッダーファイルと本文を示す。

```
// mpi_c.h
#ifndef MPI_C_H
#define MPI_C_H
// 各種定義
// データタイプの定義
typedef int MPI_Datatype;
// C++型
#define MPI_CHAR           ((MPI_Datatype)1) // 文字型
#define MPI_UNSIGNED_CHAR  ((MPI_Datatype)2) // 符号なし文字型
#define MPI_BYTE           ((MPI_Datatype)3) // バイト型
#define MPI_SHORT          ((MPI_Datatype)4) // 単精度整数型
#define MPI_UNSIGNED_SHORT ((MPI_Datatype)5) // 符号なし単精度整数型
#define MPI_INT            ((MPI_Datatype)6) // 単精度(16bit)/倍精度(32bit)整数型
#define MPI_UNSIGNED       ((MPI_Datatype)7) // 符号なし単精度(16bit)/倍精度(32bit)整数型
#define MPI_LONG           ((MPI_Datatype)8) // 倍精度整数型
#define MPI_UNSIGNED_LONG  ((MPI_Datatype)9) // 符号なし倍精度整数型
#define MPI_FLOAT          ((MPI_Datatype)10)      // 単精度浮動小数点型
#define MPI_DOUBLE         ((MPI_Datatype)11)      // 符号なし倍精度浮動小数点型
#define MPI_LONG_DOUBLE    ((MPI_Datatype)12)      // 単精度浮動小数点型
#define MPI_LONG_LONG_INT  ((MPI_Datatype)13)      // 符号なし倍精度浮動小数点型
// Fortran types
#define MPI_COMPLEX           ((MPI_Datatype)23)   // 単精度複素数型
#define MPI_DOUBLE_COMPLEX    ((MPI_Datatype)24)   // 倍精度複素数型
#define MPI_LOGICAL           ((MPI_Datatype)25)   // 論理データ型
#define MPI_REAL              ((MPI_Datatype)26)   // 単精度実数型
#define MPI_DOUBLE_PRECISION  ((MPI_Datatype)27)   // 倍精度実数型
#define MPI_INTEGER           ((MPI_Datatype)28)   // 整数型
#define MPI_2INTEGER          ((MPI_Datatype)29)   // 整数型のペア
#define MPI_2COMPLEX          ((MPI_Datatype)30)   // 複素数型のペア
#define MPI_2DOUBLE_COMPLEX   ((MPI_Datatype)31)   // 倍精度複素数型のペア
#define MPI_2REAL             ((MPI_Datatype)32)   // 単精度実数型のペア
#define MPI_2DOUBLE_PRECISION ((MPI_Datatype)33)   // 倍精度実数型のペア
#define MPI_CHARACTER         ((MPI_Datatype)34)   // 文字列型

// 構造体の定義
// 単精度複素数型 COMPLEX(COMPLEX_4)
typedef struct {
      float real;              // 実数部
      float imag;              // 虚数部
} COMPLEX, COMPLEX_4;
// 倍精度複素数型 COMPLEX_8
typedef struct {
      double real;       // 実数部
      double imag;       // 虚数部
} COMPLEX_8;
// 受信ステータス MPI_Status
typedef struct {
      int count;                    // 受信したデータの個数
      int MPI_SOURCE;               // 受信元のランク
      int MPI_TAG;                  // 受信したデータのタグ番号
      int MPI_ERROR;                // 受信の際のエラー値
```

```
} MPI_Status;
// プロセス情報     PROCESS_INFO
#ifndef MAX_COMPUTERNAME_LENGTH
#define MAX_COMPUTERNAME_LENGTH 16     // ローカルコンピュータ名の最大長さ
#endif // MAX_COMPUTERNAME_LENGTH
typedef struct {
      char   m_szHost[MAX_COMPUTERNAME_LENGTH + 1 ];      // ローカルホスト名
      int          m_nID;              // 相手のID値(ID_MASTER = 0, ID_SLAVE + n = 1+n (n=0,1,2,..))
      int          m_nRange[2];        // 担当範囲[開始 - 終了]
} PROCESS_INFO;

// 送信元の定義
#define ID_MASTER  0                   // マスター
#define      ID_SLAVE     1            // スレーブのベースランク
#define      MPI_ANY_SOURCE      -2     // ランクのワイルドカード:MPI_ANY_SOURCE

// タグの定義
#define      MPI_ANY_TAG         -1     // タグ番号のワイルドカード:MPI_ANY_TAG

// コミュニケータ
typedef int MPI_Comm;                  // コミュニケータ(現在はダミー)
#define MPI_COMM_WORLD 91              // 基本コミュニケータ：MPI_COMM_WORLD

// デフォルト受信ステータスの定義
extern MPI_Status system_stat;
#define TEMP_STATUS system_stat        // 一時使用受信ステータス:TEMP_STATUS

// 最大並列数
#define MAX_PROCESS 64                 // 最大プロセス数

// タイマー数の定義
#define MAX_TIMER 64                   // タイマーの数

// MPI関数の定義
extern "C"
{
      bool MPI_Init(int argc, char* argv[]);
      bool cMPI_Init(int nID, const char* szParam);
      bool MPI_Comm_size(MPI_Comm comm, int *nSize);
      bool MPI_Comm_rank(MPI_Comm comm, int *nRank);
      int MPI_Send(const void *pBuf, int nCount, MPI_Datatype type, int nDest,
      int nTag = MPI_ANY_TAG, MPI_Comm comm = MPI_COMM_WORLD);
      int cMPI_Send(const void *pBuf, int nSize, int nDest);
      int MPI_Bcast(void* pBuf, int nCount, MPI_Datatype type, int nRoot, MPI_Comm comm = MPI_COMM_WORLD);
      int MPI_Recv(void *pBuf, int nCount, MPI_Datatype type, int nSource,
      int nTag = MPI_ANY_TAG, MPI_Comm comm = MPI_COMM_WORLD, MPI_Status *rStatus = &TEMP_STATUS);
      int cMPI_Recv(void *pBuf, int nSize, int nSource);
      void MPI_Finalize();
      double MPI_Wtime(int num = 0);
// ユーテイリティ関数の定義
      int get_size(int nCount, MPI_Datatype type);
      int get_range_s(int nID);
      int get_range_e(int nID);
      bool time_log(const char* szFile);
```

```
}

#endif // MPI_C_H
```

次に、mpi_c.cpp を示す。

```
// mpi_c.cpp : DLL アプリケーション用のエントリ ポイントを定義します。
//
#include "stdafx.h"
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    return TRUE;
}
```

6.4 sock_func

本節では、sock_func 関数について、ヘッダーファイルと本文を示す。
最初は、ヘッダーファイル sock_func.h である。

```
// winsock関数の拡張
#ifndef SOCK_FUNC_H
#define SOCK_FUNC_H

// socket用関数
bool sock_init();                                                    // winsockの初期化を行う
void sock_close(SOCKET hSocket);                                     // ソケットを正しく切断する
SOCKET sock_connect(const char* szServer, int nPort);                // ソケットの接続を行う
SOCKET sock_listen(int nPort);                                       // ソケットの待ち受けを開始する
bool sock_accept(SOCKET hListen, SOCKET &retSocket);                 // ソケットを受け入れる
int sock_send(SOCKET hSocket, const void* lpBuf, int nSize);         // データを送信する
int sock_recv(SOCKET hSocket, void* lpBuf, int nSize);               // データを受信する
int sock_recv_check(SOCKET hSocket);                                 // 未受信のデータ量を調べる
int set_send_buf(SOCKET hSocket, int nBufSize);                      // ソケット送信バッファを拡張する
int get_send_buf(SOCKET hSocket, int *pBufSize);                     // ソケット送信バッファサイズを取得する
int set_recv_buf(SOCKET hSocket, int nBufSize);                      // ソケット受信バッファを拡張する
int get_recv_buf(SOCKET hSocket, int *pBufSize);                     // ソケット受信バッファサイズを取得する
bool get_local_IP(char* szAddress, int nSize);                       // ローカルコンピュータ名を取得する

// 各種定義
// ポートの定義
#define             PORT_DEAMON         1950              // デーモンポート
#define             PORT_PARALLEL       2000              // 基本ポート
// メッセージ
#define             MPI_MSG_BOOT        1000              // デーモンへのプロセス起動メッセージ
#define             PARAM_SIZE          50                // プロセスを起動する際のコマンドライン引数の
                                                          // 各パラメータの最大文字数
```

```
// 送信最大サイズ
#define MAX_SIZE 65535                              // 一回のsendで送信可能な最大サイズ(Byte)

#endif // SOCK_FUNC_H
```

次に、sock_func.cpp について示す。

```
// winsock拡張関数の実装
#include "stdafx.h"
#include "sock_func.h"

// ====================================================================
// winsockAPIをカプセル化した関数群
// エラーが起きた場合はWSAGetLastError()でエラー値が取得できる。
// ====================================================================

// ====================================================================
// grobal::sock_init
// 概要　: winsockの初期化を行う
// 引数　: none
// 戻り値: bool    : 初期化が成功すればtrueを返す。失敗した場合はfalseを返す
// ====================================================================
bool sock_init()
{
      int nRet;
      WORD wVersionRequested = MAKEWORD(2,0);          // WinSockバージョン2.0
      WSADATA wsaData;                                       // WinSock情報
      // WinSockの初期化
      nRet = WSAStartup(wVersionRequested, &wsaData);
      if(wsaData.wVersion != wVersionRequested)
      {
            return false;
      }
      return true;
}

// ====================================================================
// grobal::sock_close
// 概要　: ソケットを正しく切断する。まず送信を停止し、次にバッファに溜
//                まっているデータを全て受信し、切断する。
// 引数　: hSocket = 切断したいソケットのハンドル
// 戻り値: none
// ====================================================================
void sock_close(SOCKET hSocket)
{
      int nRet;
      char szBuf[1000];
      // 接続を切断する
      shutdown(hSocket, 1);          // 送信停止
      do                                          // 残っているデータを全て受信
      {
            nRet = recv(hSocket, szBuf, sizeof(szBuf), 0);
      }while(nRet > 0);
```

```
	shutdown(hSocket, 2);		// 受信停止
	closesocket(hSocket);		// 切断
}

// ================================================================
// grobal::sock_connect
// 概要　: 指定コンピュータ名(IPアドレス)の指定ポートへTCP/IPで接続を行う。
//		接続されるまで操作はブロッキングされる。
//		さらに送受信バッファをMAX_SIZE(sock_func.hで指定)だけ確保する。
// 引数　:　szServer	= 接続したい相手のコンピュータ名(IPアドレス)
//		nPort		= 接続したいポート番号
// 戻り値: SOCKET　: 接続が成功した場合はソケットのハンドルが返る。
//					接続が失敗した場合はINVALID_SOCKETが返る。
// ================================================================
SOCKET sock_connect(const char* szServer, int nPort)
{
	SOCKET hSocket;
	LPHOSTENT lpHostEntry;			// サーバアドレス
	int nRet;

	// TCP/IPソケットを作成
	hSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
	if(hSocket == INVALID_SOCKET)
	{
		return INVALID_SOCKET;
	}

	// 相手を検索
	lpHostEntry = gethostbyname(szServer);
	if(lpHostEntry == NULL)
	{
		return INVALID_SOCKET;
	}

	// アドレス構造体を埋める
	SOCKADDR_IN saServer;
	saServer.sin_family = AF_INET;
	saServer.sin_addr = *((LPIN_ADDR)*lpHostEntry->h_addr_list);
	saServer.sin_port = htons(nPort);

	// 相手と接続
	nRet = connect(hSocket, (LPSOCKADDR)&saServer, sizeof(SOCKADDR));
	if(nRet == SOCKET_ERROR)
	{
		closesocket(hSocket);
		return INVALID_SOCKET;
	}
	// 送受信バッファサイズを設定する
	nRet = set_recv_buf(hSocket, MAX_SIZE);
	nRet = set_send_buf(hSocket, MAX_SIZE);

	return hSocket;
}
```

```
// ================================================================
// grobal::sock_listen
// 概要　: 指定ポートでTCP/IPで接続の待ち受けを開始する。
// 引数　:　 nPort        = 待ち受けしたいポート番号
// 戻り値: SOCKET　: 待ち受け開始した場合はソケットのハンドルが返る。
//                                  待ち受けが失敗した場合はINVALID_SOCKETが返る。
// ================================================================
SOCKET sock_listen(int nPort)
{
      int nRet;
      SOCKET hListen;
      hListen = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
      if(hListen == INVALID_SOCKET)
      {
            return INVALID_SOCKET;
      }
      // アドレス構造体を埋める
      SOCKADDR_IN saServer;                        // サーバアドレス
      saServer.sin_family = AF_INET;
      saServer.sin_addr.s_addr = INADDR_ANY;
      saServer.sin_port = htons(nPort);

      // ソケットをアドレスに関連付ける
      nRet = bind(hListen, (LPSOCKADDR)&saServer, sizeof(SOCKADDR_IN));
      if(nRet == SOCKET_ERROR)
      {
            closesocket(hListen);
            return INVALID_SOCKET;
      }
      // クライアントからの受信を待つ
      nRet = listen(hListen, SOMAXCONN);     // WinSockにこのソケットで待ち受けすることを告げる
      if(nRet == SOCKET_ERROR)
      {
            closesocket(hListen);
            return INVALID_SOCKET;
      }
      return hListen;
}

// ================================================================
// grobal::sock_accept
// 概要　: 待ち受け処理(sock_listen)を行ったソケットに、他のPCから接続
//                  要求があった場合、受け入れ処理を行う。
//                  他のPCから接続要求があるまで操作はブロッキングされる。
//           ※　プログラマがソケットを生成する必要はなく、
//                  SOCKET型の変数を用意し、retSocketとして渡すだけでよい。
// 引数　:　 hListen            = 待ち受け処理(sock_listen)中のソケットのハンドル
//                  retSocket    = 接続要求のあったPCと接続されたソケットのハンドル。
// 戻り値: bool    : 受け入れ処理を行ったソケットの作成が失敗すればfalseを返す。
// ================================================================
bool sock_accept(SOCKET hListen, SOCKET &retSocket)
{
      int nRet;
      // 待ち受け用ソケットに接続があるまで待つ
```

```
	retSocket = accept(hListen, NULL, NULL);
	if(retSocket == INVALID_SOCKET)
	{
		closesocket(hListen);
		return false;
	}
	// 送受信バッファサイズを設定する
	nRet = set_recv_buf(retSocket, MAX_SIZE);
	nRet = set_send_buf(retSocket, MAX_SIZE);
	if(nRet == SOCKET_ERROR)  return false;
	return true;
}

// ================================================================
// grobal::sock_send
// 概要　: 接続が確立しているソケットからデータの送信を行う。
//		データの送信が完了するかエラーがあるまで操作はブロッキングされる。
//	※　データの送信の順序,サイズと相手の受信の順序,サイズが正しく一致
//		しないとデータは正しく転送されず、アプリケーションはデッドロックを
//		起こし、止まってしまうので細心の注意が必要である。
// 引数　:　hSocket	= データを送信したいソケットのハンドル
//		lpBuf	= 送信したいデータの先頭アドレス(データ型は問わない)
//		nSize	= 送信したいデータのサイズ(sizeof演算子で取得)
// 戻り値: int	: 送信したデータのサイズが返る。
//			エラーなどで途中終了した場合は途中まで送信したサイズが返る
// ================================================================
int sock_send(SOCKET hSocket, const void* lpBuf, int nSize)
{
	int nSizeSend;				// 転送バイトサイズ
	int nErr;				// エラー値
	PBYTE pBuf = (PBYTE)lpBuf;	// データのアドレス
	int nWritten;				// 送信バイト数(戻り値)
	int nLeft = nSize;			// 残りバイト数

	while(nLeft > 0)
	{
		// 転送サイズを設定
		if(nLeft > MAX_SIZE)	nSizeSend = MAX_SIZE;
		else nSizeSend = nLeft;
		// 何かを送信するか、エラーが出るまで送信を続ける
		do
		{
			nWritten = send(hSocket, (LPSTR)pBuf, nSizeSend, 0);
			// エラー処理
			if(nWritten == SOCKET_ERROR)
			{
				nErr = WSAGetLastError();
				// ブロック中ならば再送信する
				if(nErr == WSAEWOULDBLOCK || nErr == WSAENOBUFS)
				{
					continue;
				}
				return SOCKET_ERROR;
			}
```

```
            } while(nWritten == SOCKET_ERROR);
            nLeft -= nWritten;              // 残りバイト数を計算
            pBuf += nWritten;               // データの送信位置を移動
      }
      return nSize - nLeft;
}

// ===================================================================
// grobal::sock_recv
// 概要　: 接続が確立しているソケットからデータの受信を行う。
//               データの受信が完了するかエラーがあるまで操作はブロッキングされる。
//          ※　データの送信の順序,サイズと相手の受信の順序,サイズが正しく一致
//               しないとデータは正しく転送されず、アプリケーションはデッドロックを
//               起こし、止まってしまうので細心の注意が必要である。
// 引数　:　hSocket      = データを受信したいソケットのハンドル
//               lpBuf  = 受信したデータの保存先のアドレス(データ型は問わない)
//               nSize  = 受信したいデータのサイズ(sizeof演算子で取得)
// 戻り値: int     : 受信したデータのサイズが返る。
//                    エラーなどで途中終了した場合は途中まで受信したサイズが返る
// ===================================================================
int sock_recv(SOCKET hSocket, void* lpBuf, int nSize)
{
      int nRead;                            // 受信バイト数(戻り値)
      int nErr;                             // エラー値
      PBYTE pBuf = (PBYTE)lpBuf;      // 保存先アドレス
      int nLeft = nSize;              // 残りバイト数

      // バッファの初期化
      memset(pBuf, 0, nSize);
      while(nLeft > 0)
      {
            // 何かを受信するか、エラーが出るまで受信を続ける
            do
            {
                  nRead = recv(hSocket, (LPSTR)pBuf, nLeft, 0);
                  // エラー処理
                  if(nRead == SOCKET_ERROR)
                  {
                        nErr = WSAGetLastError();
                        // ブロック中ならば再受信する
                        if(nErr == WSAEWOULDBLOCK)
                        {
                              continue;
                        }
                        return SOCKET_ERROR;
                  }
            } while(nRead == SOCKET_ERROR);
            nLeft -= nRead;               // 残りバイト数を計算
            pBuf += nRead;                // データの受信位置を移動
      }
      return nSize - nLeft;
}

// ===================================================================
```

```
// grobal::sock_recv_check
// 概要　: ソケットに溜まっている未受信のデータの大きさを返す。
//                (データは届いているが、まだsock_recvを呼び出していない場合や、
//                sock_recvを呼び出して受信した以上にデータが送られてきている場合)
//          ※　この操作はブロッキングを起こすことはない。
// 引数　:　hSocket      = 未受信のデータ量をチェックしたいソケットのハンドル
// 戻り値: int     : 未受信のデータのサイズが返る。
//                      切断されている場合など、確認ができない場合は-1が返る
// ===================================================================
int sock_recv_check(SOCKET hSocket)
{
      int nRet;
      u_long nRecvSize;               // 未受信のデータサイズ
      nRet = ioctlsocket(hSocket, FIONREAD, &nRecvSize);
      if(nRet == SOCKET_ERROR)  return -1;
      return nRecvSize;
}

// ===================================================================
// grobal::set_send_buf
// 概要　: winsockが用いる送信用のバッファをメモリ上に確保する。
//                この関数を呼ばなくても8kバイト程度のメモリは確保されているが
//                set_send_bufを呼び出し、送信用バッファを拡張することで
//                転送スループットが向上する場合がある。
// 引数　:　hSocket           = 送信用バッファを拡張したいソケットのハンドル
//                nBufSize     = 確保したいバッファのサイズ
// 戻り値: int     : 失敗した場合はSOCKET_ERRORが返る
// ===================================================================
int set_send_buf(SOCKET hSocket, int nBufSize)
{
      int nRet;
      return nRet = setsockopt(hSocket, SOL_SOCKET, SO_SNDBUF, (char*)&nBufSize, sizeof(int));
}

// ===================================================================
// grobal::get_send_buf
// 概要　: winsockが用いる送信用のバッファサイズを取得する。
// 引数　:　hSocket           = 送信用バッファサイズを取得したいソケットのハンドル
//                nBufSize     = 確保されているバッファのサイズの保存先
// 戻り値: int     : 失敗した場合はSOCKET_ERRORが返る
// ===================================================================
int get_send_buf(SOCKET hSocket, int *pBufSize)
{
      int nRet;
      int nOptLen = sizeof(int);
      return nRet = getsockopt(hSocket, SOL_SOCKET, SO_SNDBUF, (char*)pBufSize, &nOptLen);
}

// ===================================================================
// grobal::set_recv_buf
// 概要　: winsockが用いる受信用のバッファをメモリ上に確保する。
//                この関数を呼ばなくても8kバイト程度のメモリは確保されているが
//                set_recv_bufを呼び出し、受信用バッファを拡張することで
//                転送スループットが向上する場合がある。
```

```
// 引数　:　hSocket            = 受信用バッファを拡張したいソケットのハンドル
//               nBufSize      = 確保したいバッファのサイズ
// 戻り値: int      : 失敗した場合はSOCKET_ERRORが返る
// ===================================================================
int set_recv_buf(SOCKET hSocket, int nBufSize)
{
      int nRet;
      return nRet = setsockopt(hSocket, SOL_SOCKET, SO_RCVBUF, (char*)&nBufSize, sizeof(int));
}

// ===================================================================
// grobal::get_recv_buf
// 概要　: winsockが用いる受信用のバッファサイズを取得する。
// 引数　:　hSocket            = 受信用バッファサイズを取得したいソケットのハンドル
//               nBufSize      = 確保されているバッファのサイズ
// 戻り値: int      : 失敗した場合はSOCKET_ERRORが返る
// ===================================================================
int get_recv_buf(SOCKET hSocket, int *pBufSize)
{
      int nRet;
      int nOptLen = sizeof(int);
      return nRet = getsockopt(hSocket, SOL_SOCKET, SO_RCVBUF, (char*)pBufSize, &nOptLen);
}

// ===================================================================
// grobal::get_local_IP
// 概要　: ローカル(自分の)コンピュータ名を取得する。
// 引数　:　szAddress    = 取得したコンピュータ名の保存先
//               nSize         = コンピュータ名の保存先のサイズ(strlen()で取得)
// 戻り値: bool     : 指定した保存先のサイズが小さい場合やコンピュータ名の
//                           取得に失敗した場合はfalseが返る
// ===================================================================
bool get_local_IP(char* szAddress, int nSize)
{
      TCHAR szLocal[MAX_COMPUTERNAME_LENGTH + 1 ];
      DWORD dwRet = MAX_COMPUTERNAME_LENGTH + 1;
      LPHOSTENT lpHostEntry;          // hostent構造体のポインタ
      struct in_addr *pInAddr;  // インターネットアドレスを指すポインタ

      if(nSize < MAX_COMPUTERNAME_LENGTH + 1)
      {
            return false;
      }
      // ホストエントリを取得
      if(!GetComputerName(szLocal, &dwRet))
      {
            return false;
      }
      lpHostEntry = gethostbyname(szLocal);
      if(lpHostEntry == NULL)
      {
            return false;
      }
      // ＩＰアドレスを取得
```

```
        pInAddr = ((LPIN_ADDR)lpHostEntry->h_addr_list[0]);

        strcpy(szAddress, inet_ntoa(*pInAddr));

        return true;
}
```

6.5 mpi_func.cpp

ここでは、mpi_func.cpp の本文を示す。

```
// MPI関数の実装
#include "stdafx.h"
#include "mpi_c.h"
#include "MPI_C_Core.h"

#include <stdlib.h>
// システムの実体
CMPI_C_Core core;
// 一時受信ステータスの実体
MPI_Status system_stat;

// ====================================================================
// MPI関数群
// DLLの外部から使用できる関数が収められている
// ====================================================================

// ====================================================================
// grobal::MPI_Init
// 概要　: mpi_cの初期化を行う。プロセス情報を元に全てのプロセスを接続を行う。
//                 使用方法としては、
//                 コマンドライン引数をそのまま引数としてMPI_Init()に渡せばよい。
//                 初期化の基本的な流れは以下の通りである。
//                 ①プロセス情報の取得
//                       master : 設定ファイルを読み込む
//                       slave  : 一つ前のプロセスと接続し、プロセス情報を受け取る。
//                 ②接続アルゴリズム
//                       自分より小さいID          : 接続する
//                       自分より一つ大きいID      : 起動させ、接続を待ち受ける
//                       自分より大きいID          : 相手からの接続を待ち受ける
//          ※  起動させるプロセスではmpi_cデーモンが起動している必要がある。
// 引数　:   argc  = コマンドライン引数の個数
//                 argv  = コマンドライン引数
// 戻り値: bool    : 実行時に引数として渡した値が２個以外であった場合や、
//                           初期化に失敗した場合はfalseが返る
// ====================================================================
bool MPI_Init(int argc, char* argv[])
{
      int nID;
      if(argc != 3)      return false;
      // 引数を取得      ※  argv[0]は実行ファイル名なので無視
      nID = atoi(argv[1]);      // IDを取得
```

```
	return core.Init(nID, argv[2]);
}

// ================================================================
// grobal::cMPI_Init
// 概要　: mpi_cの初期化を行う。プロセス情報を元に全てのプロセスを接続を行う。
//		初期化の基本的な流れは以下の通りである。
//		①プロセス情報の取得
//			master : 設定ファイルを読み込む
//			slave  : 一つ前のプロセスと接続し、プロセス情報を受け取る。
//		②接続アルゴリズム
//			自分より小さいID		: 接続する
//			自分より一つ大きいID	: 起動させ、接続を待ち受ける
//			自分より大きいID		: 相手からの接続を待ち受ける
//	※　起動させるプロセスではmpi_cデーモンが起動している必要がある。
// 引数　:　nID		= 自分のID
//		szParam		= パラメータ引数
//				master: プロセス設定ファイルのフルパス,
//				slave: 一つ前のIDのコンピュータ名(接続用に用いる)
// 戻り値: bool	: 他のプロセスとの接続に失敗した場合はfalseが返る
// ================================================================
bool cMPI_Init(int nID, const char* szParam)
{
	return core.Init(nID, szParam);
}

// ================================================================
// grobal::MPI_Comm_size
// 概要　: mpi_cで起動しているプロセスの数を取得する
// 引数　:　MPI_Comm	= 現在は未使用[ダミー]
//		nSize		= プロセス数の保存先
// 戻り値: bool	: 現在はダミーなので必ずtrueが返る
// ================================================================
bool MPI_Comm_size(MPI_Comm comm, int *nSize)
{
	*nSize = core.GetProcessCount();
	return true;
}

// ================================================================
// grobal::MPI_Comm_rank
// 概要　: mpi_c内での自分のランクを取得する
// 引数　:　MPI_Comm	= 現在は未使用[ダミー]
//		nRank		= 自分のランクの保存先
// 戻り値: bool	: 現在はダミーなので必ずtrueが返る
// ================================================================
bool MPI_Comm_rank(MPI_Comm comm, int *nRank)
{
	*nRank = core.GetMyID();
	return true;
}

// ================================================================
// grobal::MPI_Send
```

```
// 概要　: 指定ランクへのデータの送信を行う。
//                 データの送信が完了するかエラーがあるまで操作はブロッキングされる。
//           ※　データの送信の順序,型,個数と相手の受信の順序,型,個数が正しく一致
//                 しないとデータは正しく転送されず、アプリケーションはデッドロックを
//                 起こし、止まってしまうので細心の注意が必要である。
// 引数　:　 pBuf　= 送信したいデータの先頭アドレス
//                 nCount = 送信するデータの個数
//                 type　= 送信するデータの型(mpi_c.hで指定されているデータ型)
//                 nDest　= 送信先のランク
//                 nTag　= タグ番号(現在は未使用)
//                 comm　= コミュニケータ(現在は未使用)
// 戻り値: int　　: 送信したデータのサイズが返る。
//                       エラーなどで途中終了した場合は途中まで送信したサイズが返る
// ================================================================
int MPI_Send(const void *pBuf, int nCount, MPI_Datatype type, int nDest, int nTag, MPI_Comm comm)
{
      return core.Send(nDest, pBuf, get_size(nCount, type));
}

// ================================================================
// grobal::cMPI_Send
// 概要　: 指定ランクへのデータの送信を行う。
//                 データの送信が完了するかエラーがあるまで操作はブロッキングされる。
//           ※　データの送信の順序,データサイズと相手の受信の順序,データサイズが
//                 正しく一致しないとデータは正しく転送されず、アプリケーションは
//                 デッドロックを起こし、止まってしまうので細心の注意が必要である。
// 引数　:　 pBuf　= 送信したいデータの先頭アドレス
//                 nSize　= 送信したいデータのサイズ(sizeof演算子で取得)
//                 nDest　= 送信先のランク
// 戻り値: int　　: 送信したデータのサイズが返る。
//                       エラーなどで途中終了した場合は途中まで送信したサイズが返る
// ================================================================
int cMPI_Send(const void *pBuf, int nSize, int nDest)
{
      return core.Send(nDest, pBuf, nSize);
}

// ================================================================
// grobal::MPI_Bcast
// 概要　: 全てのプロセスにデータを一斉送信する。
//           ※ MPI_Bcast()を用いた場合、データの受信側でも必ず
//                 MPI_Bcast()を用いることにより受信を行わなければならない。
//                 MPI_Recvで受信を試みた場合の動作は不定であるので
//                 注意が必要である。
// 引数　:　 pBuf　= 送信データの先頭アドレス(データ型は問わない)
//                 nCount = 送信するデータの個数
//                 type　= 送信するデータの型(mpi_c.hで指定されているデータ型)
//                 nRoot　= 送信元プロセスのID
//                 comm　= コミュニケータ(現在は未使用)
// 戻り値: int　　: 一斉送信を行う側で失敗した場合、nSize以外の値が返る
//                         受信する側で失敗した場合は、途中まで受信したサイズが返る
// ================================================================
int MPI_Bcast(void* pBuf, int nCount, MPI_Datatype type, int nRoot, MPI_Comm comm)
{
```

```
	return core.BCast(nRoot, pBuf, get_size(nCount, type));
}

// ================================================================
// grobal::MPI_Recv
// 概要　: 指定ランクからデータの受信を行う。
//		データの受信が完了するかエラーがあるまで操作はブロッキングされる。
//	※　データの送信の順序,サイズと相手の受信の順序,サイズが正しく一致
//		しないとデータは正しく転送されず、アプリケーションはデッドロックを
//		起こし、止まってしまうので細心の注意が必要である。
//		受信ステータスは以下のように構成されている構造体である。
//	MPI_Status:  count	= 受信したデータの個数
//				MPI_SOURCE	= 受信元のランク
//				MPI_TAG		= 受信したデータのタグ
//				MPI_ERROR	= 受信したデータに関するエラー値
// 引数　:　pBuf	= 受信したデータの保存先のアドレス
//		nCount	= 受信するデータの個数
//		type	= 受信するデータの型(mpi_c.hで指定されているデータ型)
//		nSource	= 受信元のランク
//		nTag	= タグ番号(現在は未使用)
//		comm	= コミュニケータ(現在は未使用)
//		pStatus	= 受信ステータスの保存先
// 戻り値: int	: 受信したデータのサイズが返る。
//			エラーなどで途中終了した場合は途中まで受信したサイズが返る
// ================================================================
int MPI_Recv(void *pBuf, int nCount, MPI_Datatype type, int nSource, int nTag,
	MPI_Comm comm, MPI_Status *pStatus)
{
	int i;
	int nRet;
	int nProc = core.GetProcessCount();
	// ランクの指定がない場合
	if(nSource == MPI_ANY_SOURCE)
	{
		// selectに渡すソケット構造体をセット
		fd_set fds;
		FD_ZERO(&fds);
		for(i=0; i < nProc; i++)
		{
			FD_SET(core.m_pSocket[i], &fds);
		}
		// 受信可能になるまで待ち受け
		nRet = select(fds.fd_count+1, &fds, NULL, NULL, NULL);
		// 受信可能になったソケットを探す
		while(nSource == MPI_ANY_SOURCE)
		{
			for(i=0; i < nProc; i++)
			{
				nRet = sock_recv_check(core.m_pSocket[i]);
				if(nRet > 0)
				{
					nSource = i;	// 受信可能なソケットを指定ランクとする
					break;
				}
```

```
                    }
                }
            }
            // 指定ランクから受信する
            nRet = core.Recv(nSource, pBuf, get_size(nCount, type));
            // 受信ステータスをセット
            pStatus->count           = nCount;
            pStatus->MPI_SOURCE= nSource;
            pStatus->MPI_TAG         = nTag;
            pStatus->MPI_ERROR = nRet;

            return nRet;
    }


    // ================================================================
    // grobal::cMPI_Recv
    // 概要　: 指定ランクからデータの受信を行う。
    //                 データの受信が完了するかエラーがあるまで操作はブロッキングされる。
    //           ※　データの送信の順序,サイズと相手の受信の順序,サイズが正しく一致
    //                 しないとデータは正しく転送されず、アプリケーションはデッドロックを
    //                 起こし、止まってしまうので細心の注意が必要である。
    // 引数　:　 pBuf　= 受信したデータの保存先のアドレス
    //                 nSize　= 受信したいデータのサイズ(sizeof演算子で取得)
    //                 nSource　　= 受信元のランク
    // 戻り値: int　　: 受信したデータのサイズが返る。
    //                       エラーなどで途中終了した場合は途中まで受信したサイズが返る
    // ================================================================
    int cMPI_Recv(void *pBuf, int nSize, int nSource)
    {
            return core.Recv(nSource, pBuf, nSize);
    }

    // ================================================================
    // grobal::MPI_Finalize
    // 概要　: mpi_cの終了処理を行う。アプリケーションを終了する前に必ず呼び出す
    //                 必要がある。
    // 引数　:　 none
    // 戻り値: none
    // ================================================================
    void MPI_Finalize()
    {
            core.ShutDownAll();
    }

    // ================================================================
    // grobal::MPI_Wtime
    // 概要　: 時間の計測を行う。最初の呼び出しで計測を開始し、
    //                 二度目の呼び出しで計測を終了し、計測した時間をマイクロ秒単位で返す。
    //                 独立したタイマーを保持しているので
    //                 MAX_TIMER(mpi_c.hで定義)個を同時に計測することができる。
    // 引数　:　 num　= 用いるタイマーの番号(0からMAX_TIMER-1までの間)
    // 戻り値: double　: 二度目の呼び出しでは計測された時間(マイクロ秒)を返す。
    //                                   一度目の呼び出しでは0,エラーの場合は-1を返す
```

```
// ================================================================
double MPI_Wtime(int num)
{
	double nRet;
	static bool s_bStat[MAX_TIMER] = {false};		// 開始／終了フラグ
	s_bStat[num] = s_bStat[num] ? false : true;  // 開始／終了フラグのON・OFFを切り替える

	if(s_bStat[num] == true)
	{
		nRet = core.Time_Start(num);
	}
	else
	{
		nRet = core.Time_Stop(num);
	}
	return nRet;
}

// ================================================================
// grobal::get_size
// 概要　: MPIデータ型と個数から必要とするバッファサイズを取得する。
// 引数　:　nCount = データの個数
//		type　= MPIデータ型(mpi_c.hで指定されているデータ型)
// 戻り値: int	: 必要とするバッファサイズを返す。
//				無効なデータ型の場合は-1を返す
// ================================================================
int get_size(int nCount, MPI_Datatype type)
{
	if(nCount <= 0)	return -1;
	switch(type)
	{
	case MPI_CHAR:
		return sizeof(char) * nCount;
	case MPI_UNSIGNED_CHAR:  case MPI_CHARACTER:
		return sizeof(unsigned char) * nCount;
	case MPI_BYTE:
		return sizeof(unsigned char) * nCount;
	case MPI_SHORT:
		return sizeof(short) * nCount;
	case MPI_UNSIGNED_SHORT:
		return sizeof(unsigned short) * nCount;
	case MPI_INT:	case MPI_LOGICAL: case MPI_INTEGER:
		return sizeof(int) * nCount;
	case MPI_UNSIGNED:
		return sizeof(unsigned) * nCount;
	case MPI_LONG:
		return sizeof(long) * nCount;
	case MPI_UNSIGNED_LONG:
		return sizeof(unsigned long) * nCount;
	case MPI_FLOAT:	case MPI_REAL:
		return sizeof(float) * nCount;
	case MPI_DOUBLE:	case MPI_DOUBLE_PRECISION:
		return sizeof(double) * nCount;
	case MPI_LONG_DOUBLE:
```

```
            return sizeof(long double) * nCount;
        case MPI_COMPLEX:
            return sizeof(COMPLEX) * nCount;
        case MPI_DOUBLE_COMPLEX:
            return sizeof(COMPLEX_8) * nCount;
        default:
            return -1;
        }
}

// ================================================================
// grobal::get_range_s
// 概要　: 担当範囲(開始)を取得する
// 引数　:　 nID　= 担当範囲を取得したいプロセスのID
// 戻り値: int　　: 担当範囲(開始)を返す
// ================================================================
int get_range_s(int nID)
{
    return core.GetRange_S(nID);
}

// ================================================================
// grobal::get_range_e
// 概要　: 担当範囲(終了)を取得する
// 引数　:　 nID　= 担当範囲を取得したいプロセスのID
// 戻り値: int　　: 担当範囲(終了)を返す
// ================================================================
int get_range_e(int nID)
{
    return core.GetRange_E(nID);
}

// ================================================================
// grobal::time_log
// 概要　: 指定されたファイル名にCPUカウンタ周波数と
//                各タイマーのカウンタ値を記録する。
// 引数　:　 szFile= 記録するファイル名
// 戻り値: bool　 : ファイルのオープンに失敗すればfalseを返す
// ================================================================
bool time_log(const char* szFile)
{
    return core.TimeLog(szFile);
}
```

6.6 その他のプログラム

ここでは、ライブラリー内で使用されていて、しかも、これまでに掲載されていないプログラム及び定義文の内容を以下に示す。

```
; mpi_c.def : DLL 用のモジュール パラメータ宣言

LIBRARY      "mpi_c"
DESCRIPTION  'mpi_c Windows Dynamic Link Library'

EXPORTS
      MPI_Init           @1
      cMPI_Init          @2
      MPI_Send           @3
      cMPI_Send          @4
      MPI_Recv           @5
      cMPI_Recv          @6
      MPI_Finalize @7
      get_size           @8
      MPI_Comm_size      @9
      MPI_Comm_rank      @10
      MPI_Bcast          @11
      MPI_Wtime          @12
      get_range_s        @13
      get_range_e        @14
      time_log           @15


C     ―――――――――――――――――――――――――――――――――
C     ●   MODULE / MPI_DEFINE
C     ―――――――――――――――――――――――――――――――――
C     ●   mpi_c 関数を Fortran から用いるための参照モジュール
C     ―――――――――――――――――――――――――――――――――
      MODULE MPI_DEFINE
      implicit none
      INTEGER*4, PARAMETER ::               ! fortran : C++
     &           MPI_COMPLEX= 23,           ! COMPLEX : struct COMPLEX
     &           MPI_DOUBLE_COMPLEX= 24,    ! DOUBLE_COMPLEX  : struct COMPLEX_8
     &           MPI_LOGICAL= 25,           ! LOGICAL = int
     &           MPI_REAL = 26,             ! REAL : double
     &           MPI_DOUBLE_PRECISION = 27,! DOUBLE_PRECISION   : double
     &           MPI_DOUBLE= 27,
     &           MPI_INTEGER= 28,           ! INT : int
     &           MPI_CHARACTER= 33          ! CHARCTER = unsinged char

!タグの定義
      INTEGER*4, PARAMETER ::MPI_ANY_TAG     = -1
!ID の定義
      INTEGER*4, PARAMETER ::ID_MASTER       = 0,
     &                       ID_SLAVE        = 1,
     &                       MPI_ANY_SOURCE= -2      ! 現在は使用不可能
!   コミュニケータの定義
      INTEGER*4, PARAMETER :: MPI_COMM_WORLD = 91
!  MPI_STATUS のサイズ
      INTEGER*4, PARAMETER :: MPI_STATUS_SIZE = 4
!  MPI_STATUS の番号
      INTEGER*4, PARAMETER :: MPI_STATUS_COUNT= 1
      INTEGER*4, PARAMETER :: MPI_STATUS_SOURCE= 2
      INTEGER*4, PARAMETER :: MPI_STATUS_TAG   = 3
```

```
      INTEGER*4, PARAMETER :: MPI_STATUS_ERROR= 4

!――――――――――――――――――――――――――――――――――――――――
!                        ※　拡張するときは、以下に記してください。
!――――――――――――――――――――――――――――――――――――――――

c――――――――――――――――――――――――――――――――★構造体宣言
C     ――――――――――――――――――――――――――――――――――――
C     ●   mystatus 構造体
C     ――――――――――――――――――――――――――――――――――――
      TYPE mystatus
      integer    Rank                  !  自分の MPI の Rank
      integer    GroupCount            !  MPI 全体の数
      end TYPE
C     ――――――――――――――――――――――――――――――――――――
C     ●   MPI_STATUS 構造体
C     ――――――――――――――――――――――――――――――――――――
      TYPE  MPI_STATUS
      integer nCount;                  ! 受信データの個数
      integer nSource;                 ! 送信元のランク
      integer nTag;                    ! 受信データのタグ
      integer nErr;                    ! エラーコード
      end TYPE
      END MODULE
C     ――――――――――――――――――――――――――――――――――――


C     ――――――――――――――――――――――――――――――――――――
C     ●   MODULE / MPI_C_Func
C     ――――――――――――――――――――――――――――――――――――
C     ●   mpi_c 関数を Fortran から用いるための参照モジュール
C     ――――――――――――――――――――――――――――――――――――
      MODULE MPI_C_FUNC
C     ――――――――――――――――――――――――――――――――――――
C     ●   cMPI_Init
C     ――――――――――――――――――――――――――――――――――――
      INTERFACE
      INTEGER*4 FUNCTION cMPI_Init(nID, szIP)
!DEC$ ATTRIBUTES C, ALIAS:'_cMPI_Init' :: cMPI_Init
      INTEGER*4 :: nID[VALUE]
      CHARACTER*(*) :: szIP[REFERENCE]
      END FUNCTION
      END INTERFACE
C     ――――――――――――――――――――――――――――――――――――
C     ●   MPI_Finalize
C     ――――――――――――――――――――――――――――――――――――
      INTERFACE
      SUBROUTINE MPI_Finalize()
!DEC$ ATTRIBUTES C, ALIAS:'_MPI_Finalize' :: MPI_Finalize
      END SUBROUTINE
      END INTERFACE
C     ――――――――――――――――――――――――――――――――――――
C     ●   cMPI_Comm_size
C     ――――――――――――――――――――――――――――――――――――
```

```
      INTERFACE
      INTEGER*4 FUNCTION cMPI_Comm_size(comm, nSize)
!DEC$ ATTRIBUTES C, ALIAS:'_MPI_Comm_size' :: cMPI_Comm_size
      INTEGER*4 :: comm[VALUE]
      INTEGER*4 :: nSize[REFERENCE]
      END FUNCTION
      END INTERFACE
C     ―――――――――――――――――――――――――――――――――
C     ●   cMPI_Comm_rank
C     ―――――――――――――――――――――――――――――――――
      INTERFACE
      INTEGER*4 FUNCTION cMPI_Comm_rank(comm, nRank)
!DEC$ ATTRIBUTES C, ALIAS:'_MPI_Comm_rank' :: cMPI_Comm_rank
      INTEGER*4 :: comm[VALUE]
      INTEGER*4 :: nRank[REFERENCE]
      END FUNCTION
      END INTERFACE
C     ―――――――――――――――――――――――――――――――――
C     ●   cMPI_Send
C     ―――――――――――――――――――――――――――――――――
      INTERFACE
      INTEGER*4 FUNCTION cMPI_Send
     &                   (pBuf, nCount, nType, nDest, nTag, comm)
!DEC$ ATTRIBUTES C, ALIAS:'_MPI_Send' :: cMPI_Send
      INTEGER*4 :: pBuf[VALUE]
      INTEGER*4 :: nCount[VALUE]
      INTEGER*4 :: nType[VALUE]
      INTEGER*4 :: nDest[VALUE]
      INTEGER*4 :: nTag[VALUE]
      INTEGER*4 :: comm[VALUE]
      END FUNCTION
      END INTERFACE
C     ―――――――――――――――――――――――――――――――――
C     ●   cMPI_Bcast
C     ―――――――――――――――――――――――――――――――――
      INTERFACE
      INTEGER*4 FUNCTION cMPI_Bcast
     &               (pBuf, nCount, nType, nRoot, comm)
!DEC$ ATTRIBUTES C, ALIAS:'_MPI_Bcast' :: cMPI_Bcast
      INTEGER*4 :: pBuf[VALUE]
      INTEGER*4 :: nCount[VALUE]
      INTEGER*4 :: nType[VALUE]
      INTEGER*4 :: nRoot[VALUE]
      INTEGER*4 :: comm[VALUE]
      END FUNCTION
      END INTERFACE
C     ―――――――――――――――――――――――――――――――――
C     ●   cMPI_Recv
C     ―――――――――――――――――――――――――――――――――
      INTERFACE
      INTEGER*4 FUNCTION  cMPI_Recv
     &               (pBuf, nCount, nType, nSource, nTag, comm, pStatus)
!DEC$ ATTRIBUTES C, ALIAS:'_MPI_Recv' :: cMPI_Recv
      INTEGER*4 :: pBuf[VALUE]
```

```
      INTEGER*4 :: nCount[VALUE]
      INTEGER*4 :: nType[VALUE]
      INTEGER*4 :: nSource[VALUE]
      INTEGER*4 :: nTag[VALUE]
      INTEGER*4 :: comm[VALUE]
      INTEGER*4 :: pStatus[VALUE]
      END FUNCTION
      END INTERFACE
C     ―――――――――――――――――――――――――――――――――
C     ●   cGetRange_s
C     ―――――――――――――――――――――――――――――――――
      INTERFACE
      INTEGER*4 FUNCTION cGetRange_s(nID)
!DEC$ ATTRIBUTES C, ALIAS:'_get_range_s' :: cGetRange_s
      INTEGER*4 :: nID[VALUE]
      END FUNCTION
      END INTERFACE
C     ―――――――――――――――――――――――――――――――――
C     ●   cGetRange_e
C     ―――――――――――――――――――――――――――――――――
      INTERFACE
      INTEGER*4 FUNCTION cGetRange_e(nID)
      !DEC$ ATTRIBUTES C, ALIAS:'_get_range_e' :: cGetRange_e
      INTEGER*4 :: nID[VALUE]
      END FUNCTION
      END INTERFACE
C     ―――――――――――――――――――――――――――――――――
C     ●   cMPI_WTime
C     ―――――――――――――――――――――――――――――――――
      INTERFACE
      REAL*8 FUNCTION cMPI_WTime(num)
      !DEC$ ATTRIBUTES C, ALIAS:'_MPI_Wtime' :: cMPI_Wtime
      INTEGER*4 :: num[VALUE]
      END FUNCTION
      END INTERFACE
C     ―――――――――――――――――――――――――――――――――
C     ●   cGetSize
C     ―――――――――――――――――――――――――――――――――
      INTERFACE
      INTEGER*4 FUNCTION cGetSize(nCount, nType)
!DEC$ ATTRIBUTES C, ALIAS:'_get_size' :: cGetSize
      INTEGER*4 :: nCount[VALUE]
      INTEGER*4 :: nType[VALUE]
      END FUNCTION
      END INTERFACE
      END MODULE
C     ―――――――――――――――――――――――――――――――――


C     ―――――――――――――――――――――――――――――――――
C     ●   MODULE / MPI_GENERIC_FUNC
C     ―――――――――――――――――――――――――――――――――
C     ●   void*型を含む C++関数を取り扱うための総称関数モジュール
C     ―――――――――――――――――――――――――――――――――
```

```
      MODULE MPI_GENERIC_FUNC
      INTERFACE
C     ―――――――――――――――――――――――――――――――――
C     ●   SUBROUTINE / MPI_Send
C     ―――――――――――――――――――――――――――――――――
      SUBROUTINE MPI_Send
     &                 (buf, nCount, nType, nDest, nTag, nComm, nErr)
!DEC$ ATTRIBUTES NO_ARG_CHECK :: buf
c 引数の宣言
      INTEGER:: buf
      INTEGER:: nCount
      INTEGER:: nType
      INTEGER:: nDest
      INTEGER:: nTag
      INTEGER:: nComm
      INTEGER:: nErr
      END SUBROUTINE

C     ―――――――――――――――――――――――――――――――――
c                              ●   SUBROUTINE / MPI_Bcast
C     ―――――――――――――――――――――――――――――――――
      SUBROUTINE MPI_Bcast
     &                   (buf, nCount, nType, nRoot, nComm, nErr)
!DEC$ ATTRIBUTES NO_ARG_CHECK :: buf
c 引数の宣言
      INTEGER:: buf
      INTEGER:: nCount
      INTEGER:: nType
      INTEGER:: nRoot
      INTEGER:: nComm
      INTEGER:: nErr
      END SUBROUTINE

C     ―――――――――――――――――――――――――――――――――
c                              ●   SUBROUTINE / MPI_Recv
C     ―――――――――――――――――――――――――――――――――
      SUBROUTINE MPI_Recv
     &             (buf, nCount, nType, nDest, nTag, nComm, status, nErr)
!DEC$ ATTRIBUTES NO_ARG_CHECK :: buf
c 引数の宣言
      INTEGER:: buf
      INTEGER:: nCount
      INTEGER:: nType
      INTEGER:: nDest
      INTEGER:: nTag
      INTEGER:: nComm
      INTEGER:: status(5)
      INTEGER:: nErr
      END SUBROUTINE
      END INTERFACE
      END MODULE
C     ―――――――――――――――――――――――――――――――――
```

```
C     ―――――――――――――――――――――――――――――――――――
C     ●   MODULE / time_module
C     ―――――――――――――――――――――――――――――――――――
C     ●   時間を計測するためのパラメータ,変数定義
C     ―――――――――――――――――――――――――――――――――――
      MODULE TIME_MODULE
      implicit none

      integer, parameter::  TIME_PRE= 1              ! 予備計算
      integer, parameter::  TIME_K  = 2              ! 係数行列作成
      integer, parameter::  TIME_NEWMARK= 3          ! Newmarkβ法
      integer, parameter::  TIME_LD = 4              ! 右辺項作成
      integer, parameter::  TIME_NONLINER= 5         ! 非線形項作成
      integer, parameter::  TIME_CAL= 6              ! 振動方程式を解く
      integer, parameter::  TIME_DISP_VEL= 7         ! 変位、速度を計算
      integer, parameter::  TIME_STRESS= 8           ! 応力の計算
      integer, parameter::  TIME_F_STRESS= 9         ! ファイバー応力の計算
      integer, parameter::  TIME_KT= 10              ! 接線剛性の計算
      integer, parameter::  TIME_NET_LD= 11          ! 右辺項の通信
      integer, parameter::  TIME_NET_ACC= 12         ! 予測加速度の通信
      integer, parameter::  TIME_NET_CONV= 13        ! 収束コードの通信
      integer, parameter::  TIME_NET_FORCE= 14       ! 部材節点力の通信
      integer, parameter::  TIME_STEP= 15            ! ステップ毎の通信
      integer, parameter::  TIME_ALL= 16             ! 総解析時間

      real*8::  dTime_PRE                            ! 予備計算
      real*8::  dTime_K                              ! 係数行列作成
      real*8::  dTime_NEWMARK                        ! Newmarkβ法
      real*8::  dTime_LD                             ! 右辺項作成
      real*8::  dTime_NONLINER                       ! 非線形項作成
      real*8::  dTime_CAL                            ! 振動方程式を解く
      real*8::  dTime_DISP_VEL                       ! 変位、速度を計算
      real*8::  dTime_STRESS                         ! 応力の計算
      real*8::  dTime_F_STRESS                       ! ファイバー応力の計算
      real*8::  dTime_KT                             ! 接線剛性の計算
      real*8::  dTime_NET_LD                         ! 右辺項の通信
      real*8::  dTime_NET_ACC                        ! 予測加速度の通信
      real*8::  dTime_NET_CONV                       ! 収束コードの通信
      real*8::  dTime_NET_FORCE                      ! 部材節点力の通信
      real*8::  dTime_STEP                           ! ステップ毎の通信

      real*8, save:: All_Time_PRE                    ! 予備計算
      real*8, save:: All_Time_K                      ! 係数行列作成
      real*8, save:: All_Time_NEWMARK                ! Newmarkβ法
      real*8, save:: All_Time_LD                     ! 右辺項作成
      real*8, save:: All_Time_NONLINER               ! 非線形項作成
      real*8, save:: All_Time_CAL                    ! 振動方程式を解く
      real*8, save:: All_Time_DISP_VEL               ! 変位、速度を計算
      real*8, save:: All_Time_STRESS                 ! 応力の計算
      real*8, save:: All_Time_F_STRESS               ! ファイバー応力の計算
      real*8, save:: All_Time_KT                     ! 接線剛性の計算
      real*8, save:: All_Time_NET_LD                 ! 右辺項の通信
      real*8, save:: All_Time_NET_ACC                ! 予測加速度の通信
      real*8, save:: All_Time_NET_CONV               ! 収束コードの通信
```

```
      real*8, save:: All_Time_NET_FORCE              ! 部材節点力の通信
      real*8, save:: All_Time_STEP                   ! ステップ毎の通信
      real*8, save:: All_Time_ALL                    ! 総解析時間

      END MODULE
```